**Software Quality Requirements**

[Without Classes](#_rrjpk2e11b2o)

[1. Main Function](#_q5q5cmkmw3ni)

[1.1 Does the program include a user-defined parameterless function called main?](#_mf8oq688lqe7)

[1.2 Are all import statements defined globally?](#_ocfydzqetvy2)

[1.3 There should not be any global identifiers in the program except for module names that are imported. All other identifiers must be initialized inside a function. One exception to this rule would be global constants.](#_h2clmw15zx1g)

[2. Comments](#_54immw8b1zx)

[2.1 Program Comment (Docstring)](#_bwaavpo08hac)

[2.2 Block Comment](#_y1zbec48skwa)

[2.3 Inline Comment](#_kqpdm9ijq3fk)

[2.4 User-defined Function Comment (Docstring)](#_ul067to5g2ff)

[3. Names](#_xz8tt4ol93h)

[3.1 Are the names descriptive?](#_c06vun79ydzl)

[3.2 Does each identifier, other than user-defined class names and global constants use lower\_case\_naming rather than other formats like camelCaseNaming?](#_gzfvui8eirq7)

[4. Repetition](#_pw2wa0xc4778)

[4.1 Have all adjacent duplicate line groups been replaced by repetition?](#_50olgb1x4wxs)

[5. Limiting Literals](#_33xl4ba6b68e)

[5.1 Does every literal (other than 0, 1, 2, -1, 0.0 and empty strings '') appear exactly once?](#_nvv5s9ijf7ph)

[5.2 Does every literal appear in an assignment statement near the start of a block or function?](#_k515cq5mxn6m)

[6. User-defined Functions](#_9ynlyxax21fh)

[6.1 Have all non-adjacent duplicate line groups been replaced by user-defined functions?](#_n2p6ietz9fh)

[6.2 Is each logical task in a separate user-defined function that can be described by a sentence with 1 or 2 verbs?](#_khoye787pnq2)

[6.3 Does each user-defined function have 5 or fewer arguments?](#_uywhp1h4232f)

[6.4 Does each user-defined function have 12 or fewer statements?](#_e59dsz50gnfn)

[6.5 Do all user-defined functions have their definition existing outside of other user-defined functions?](#_g1z4cw8kklj2)

[With Classes](#_ai2iibas5str)

[7. User-defined Classes](#_eudd0ql3cruj)

[7.1 Class Comment](#_w1enqn9mcs54)

[7.2 Does each class name use CamelCaseNaming?](#_z2iuvv6wq8j5)

[7.3 Does a user-defined class represent each group of objects that is used in two or more tasks and forms a single conceptual object?](#_asm88gyz85mt)

[7.4 Is each instance attribute of a user-defined class only used inside the class definition?](#_7xuh414vb0gn)

[7.5 Does each method in the class satisfy the software quality tests for user-defined functions?](#_psinmgw0imnw)

## **Without Classes**

## **1. Main Function**

### **1.1 Does the program include a user-defined parameterless function called main?**

The main function is special in that it acts as the starting point of a program.

| **def** **main**():  *# code that does something* |
| --- |

### **1.2 Are all import statements defined globally?**

Import statements shouldn't be within any user-defined functions, in other words, they should be defined globally, just below the program comments at the top of the code.

Like this:

| """ Program comments """ **import** random  **def** **main**():  *# code that does something*   main() |
| --- |

**Not** like this:

| """ Program comments """  **def** **main**():  **import** random  *# code that does something*  main() |
| --- |

### **1.3 There should not be any global identifiers in the program except for module names that are imported. All other identifiers must be initialized inside a function. One exception to this rule would be global constants.**

Global variables may cause your program to behave strangely due to many factors, so avoid defining any globally.

Like this:

| """ Program comments """  **import** random  **def** **main**():  my\_variable = 'hello'  *# code that does something*  main() |
| --- |

**Not** like this:

| """ Program comments """  **import** random my\_variable = 'hello'  **def** **main**():  *# code that does something*  main() |
| --- |

However, global constants can be used. Global constants are global variables, but they hold values that should **never be changed**, and have a slightly different naming convention. The names of all global constants follow the same rules as identifiers, *but* it should also be fully capitalized. Additionally, global constants should be defined at the beginning of the code, right after the import statements.

Note that in Python, there are no variables that hold an immutable constant value, and Python itself does not enforce the fact that global constants shouldn’t change. In other words, variables that are supposed to be global constants in Python can still have their value changed. The naming convention used for global constants is only used to indicate that the value held by this variable never changes.

Like this:

| """ Program comments """  **import** random  A\_GLOBAL\_CONSTANT = 'This value should never be changed' DAYS\_IN\_A\_WEEK = 7  **def** **main**():  *# code that does something*  main() |
| --- |

Using global constants can help improve the readability of your code, and future maintenance of the code. For example, let's say you've coded a program that calculates the pay for each employee. The pay rate can be a global constant since it doesn't change between employees, and can also be easily updated by changing one line of code if the pay rate does change. The global constant can also now be used whenever the pay rate is required, which gives the reader more information about the role and purpose of the literal when compared to simply writing the value of the literal.

| """ Program comments """  **import** random  PAYRATE = 19  **def** **main**():   *# hours worked by John*  john\_hours = 30    *# DO THIS!*  *# this gives more information to the reader*  print(f"John's pay is: {john\_hours \* PAYRATE}")    *# NOT THIS!*  *# the reader may not understand what the int 19 represents*  print(f"John's pay is: {john\_hours \* 19}")  main() |
| --- |

## **2. Comments**

### **2.1 Program Comment (Docstring)**

Program comments (docstrings) should:

* explain what the code is going to do, or its purpose
* appear at the beginning of the program or a user-defined function
* be formatted as a docstring (using """ or ''')

It is a good place to put other details such as who wrote the code, when it was written, and any other important details the reader should know beforehand.

| """ This is a sample program comment. Calculates the total amount of calories when given a food receipt Author: Kirby When: June 24, 2022 """  **def** **main**():  """  This is a sample function comment  """  do\_something()    main() |
| --- |

### **2.2 Block Comment**

Block comments should:

* be on its own line
* explain a single block of code
* be indented to the same level as the next line of code
* start with a # and a single space

A block of code is a grouping of lines of code that serve the same purpose, or have the same goal. A block comment is used to explain the purpose of those lines of code. This way the reader will be able to understand what's happening in a glance, without having to read through all the code.

Block comments are also great for notes for yourself, or the reader. They can be used as a reminder when coding, for planning out your program, or even to explain something that might be confusing for the reader.

| """ Program comments """  **def** **main**():  *# change prices of bread*  breads = {'baguette':5, 'croissant':2, 'brioche':4}  ...   main() |
| --- |

### **2.3 Inline Comment**

Inline comments should:

* be used sparingly
* separated by at least 2 spaces from the code
* describes the *intent* of the code

These are great for explaining or giving extra context about a specific line of code. *Do not repeat what the code is doing*, but rather explain why the code is the way it is, or why you have decided to write the code this way. Use this when you want to explain a single line of code that may be particularly complex, or is confusing to the reader.

Like this:

| """ Program comments """  **def** **main**():    ...  **for** bread **in** breads.keys():  breads[bread] = breads[bread] \* 1.05 *# price increase by 5%*   main() |
| --- |

**Not** like this:

| """ Program comments """  **def** **main**():    *# change prices of bread*  breads = {'baguette':5, 'croissant':2, 'brioche':4} *# all the breads*  **for** bread **in** breads.keys(): *# going through each bread*  breads[bread] = breads[bread] \* 1.05 *# multiply by 105%* main() |
| --- |

### **2.4 User-defined Function Comment (Docstring)**

User-defined function comments should:

* appear at the start of each user-defined function
* describe what the function does
* be indented one level from the function definition line
* have a line to describe the function of each parameter (its role and type); function annotations can also be used to describe the type of the parameters

This type of comment is used to explain functions that you create. They should make it easy for the reader to understand why this function exists, what it does, and everything that's required for it to work properly.

| **def** **add**(num1, num2):  """  Adds 2 numbers (num1 and num2) together  Returns an int  - num1: one of the numbers to add, type int  - num2: one of the numbers to add, type int  """    result = num1 + num2  **return** result |
| --- |

## **3. Names**

### **3.1 Are the names descriptive?**

Names for identifiers, functions, and so on should be self explanatory, so that any reader should be able to easily understand its purpose.

Like this:

| **def** **main**():  price = 800  tax = 0.05  total\_price = price + price \* tax  print(total\_price)  main() |
| --- |

**Not** like this:

| **def** **main**():  x = 800  y = 0.05  z = x + x \* y  print(z)  main() |
| --- |

### **3.2 Does each identifier, other than user-defined class names and global constants use lower\_case\_naming or camelCaseNaming?**

Sometimes we may want the variable name to be composed of more than one word. There are a few ways to do this: by using the lower\_case\_naming, or the camelCaseNaming. The lower\_case\_naming convention works by connecting words using underscores, and the camelCaseNaming capitalizes the first letter of each word in the variable name, excluding the first letter of the first word.

For this course, you can use either lower\_case\_naming or camelCaseNaming in your programs. Choose one convention and use it consistently throughout your program.

Like this:

| this\_is\_my\_list = ['hello', 'world'] a\_string = "hello world!" |
| --- |

**OR**

| thisIsMyList = ['hello', 'world'] aString = "hello world!" |
| --- |

## **4. Repetition**

### **4.1 Have all adjacent duplicate line groups been replaced by repetition?**

By avoiding adjacent duplicate lines, we can keep our code easily readable, and manageable. Having duplicate lines often means that when changes need to be done in the future, all these lines will have to be changed manually. Instead, loops can be used for any groups of lines that need to be repeated. Additionally, this will also save us time from having to write all those lines of code!

Like this:

| message = ['This', 'is', 'the', 'better', 'approach!'] **for** word **in** message:  print(word) |
| --- |

**Not** like this:

| **print**('There') **print**('are') **print**('so') **print**('many') **print**('adjacent') **print**('duplicate') **print**('lines!') |
| --- |

## **5. Limiting Literals**

### **5.1 Does every literal (other than 0, 1, 2, -1, 0.0 and empty strings '') appear exactly once?**

Any literal that is not listed above should be stored in a variable, which can then be used where needed. This way the reader can understand the purpose of the literal, and it'll also make it easier to make changes since only one line of code needs to be updated.

Like this:

| *# temperature in Celcius* temp\_c = 40  *# temperature in Fahrenheit* temp\_f = temp\_c \* (9/5) + 32  print('The temperature is', temp\_c, 'in Celcius, and', temp\_f, 'in Fahrenheit') |
| --- |

**Not** like this:

| *# temperature in Fahrenheit* temp\_f = 40 \* (9/5) + 32  print('The temperature is', 40, 'in Celcius, and', temp\_f, 'in Fahrenheit') |
| --- |

### **5.2 Does every literal appear in an assignment statement near the start of a block or function?**

When defining literals that will be used later, define it either right before the block of code it'll be used in, or at the beginning of the function. Remember that global variables aren't allowed ([see 1.3 above](#_h2clmw15zx1g))!

Like this:

| **def** **main**():   fruits = ['apples', 'pears', 'watermelon']  name = 'Sam'  **for** fruit **in** fruits:  print(name, 'likes', fruit) |
| --- |

**Not** like this:

| fruits = ['apples', 'pears', 'watermelon'] **def** **main**():   **for** fruit **in** fruits:  name = 'Sam'  print(name, 'likes', fruit) |
| --- |

## **6. User-defined Functions**

### **6.1 Have all *non-adjacent duplicate line groups* been replaced by user-defined functions?**

Non-adjacent duplicate line groups are similar to adjacent duplicate line groups, except this time the blocks of code are repeated in different areas throughout your program. Likewise, by avoiding these duplicate line groups, we can improve readability, and the maintenance of the code. For non-adjacent duplicate line groups, we'll use user-defined functions to avoid repeatedly writing the same lines of code, and so that we'll only need to call a single function when needed.

For example, in the following code, we want to convert the temperatures in these lists from celsius to fahrenheit, and print the results after each conversion.

| **def** **main**():  *# temperatures of different cities in Celsius*  edmonton\_temp = [22, 17, 11, 1]  calgary\_temp = [22, 16, 12, 3]    *# convert Edmonton temperatures to Fahrenheit*  **for** temp **in** edmonton\_temp:  temp\_f = temp \* (9/5) + 32  print(temp\_f)    *# convert Calgary temperatures to Fahrenheit*  **for** temp **in** calgary\_temp:  temp\_f = temp \* (9/5) + 32  print(temp\_f) |
| --- |

Since the conversion happens on multiple lines, it is more efficient to create a user defined function for this, so that each time a conversion needs to happen, we can simply call the function when needed. This also makes it much clearer to the reader what the code does since the function name is more descriptive than a math equation.

| **def** **temp\_fahrenheit**(temp):  """  Converts the given temperature (in Celsius) to Farenheit  Returns an int  - temp: the temperature to convert, type int  """  temp\_f = temp \* (9/5) + 32  **return** temp\_f  **def** **main**():  *# temperatures of different cities in Celsius*  edmonton\_temp = [22, 17, 11, 1]  calgary\_temp = [22, 16, 12, 3]    *# convert Edmonton temperatures to Fahrenheit*  **for** temp **in** edmonton\_temp:  temp\_f = temp\_fahrenheit(temp)  print(temp\_f)    *# convert Calgary temperatures to Fahrenheit*  **for** temp **in** calgary\_temp:  temp\_f = temp\_fahrenheit(temp)  print(temp\_f) |
| --- |

### **6.2 Is each *logical task* in a separate user-defined function that can be described by a sentence with 1 or 2 verbs?**

Think of a single logical task as a goal or purpose of a user-defined function. A single user-defined function should only have one goal, such that its logical task can easily be described in a sentence by using 1 or 2 verbs.

Let's use a factory as an example. To maximize efficiency, each machine in the factory has one job. There will be several machines that each create their own specific part, a machine for assembling the parts, and one to package the finished product. Similarly, user-defined functions should each complete one task in a program.

### **6.3 Does each user-defined function have 5 or fewer *arguments*?**

User-defined functions shouldn't require more than 5 arguments to work properly. If you find that the function needs more than 5 arguments, then the function may be doing too much, and should likely be split into multiple functions.

### **6.4 Does each user-defined function have 12 or fewer *statements*?**

Just like 6.3, if the user-defined function ends up having more than 12 lines (lines of code, excluding docstrings, comments and blank lines), the function may likely be taking on too many tasks. The limit here is not set in stone, rather it is a guide. Sometimes a user-defined function may have more lines, but it is still only completing a single logical task. The key point here is that the lines of code in the user-defined function should all work towards a single goal or task.

### **6.5 Do all user-defined functions have their definition existing outside of other user-defined functions?**

User-defined functions shouldn't be defined in another user-defined function, or in the main function (even though Python allows that).

Like this:

| **def** **main**():  *# code that does something*  my\_function()  **def** **my\_function**():  *# code that does something*   main() |
| --- |

**Not** like this:

| **def** **main**():  *# code that does something*  **def** **my\_function**():  *# code that does something*  my\_function()   main() |
| --- |

## **With Classes**

## 

## **7. User-defined Classes**

### **7.1 Class Comment**

Class comments should:

* Appear at the start of the class
* Indicate what the class is used for

Class comments are similar to user-defined comments in the sense that it should explain to the reader the purpose of the class. Any additional information about the class can also be placed here.

| **class** **Car**:  """  An object in this class represents a single car  """    **def** **\_\_init\_\_**(self):  """  Creates and returns a car object  - self: the Car to initialize  """ |
| --- |

### **7.2 Does each class name use CamelCaseNaming?**

Just like the identifiers, the naming convention you choose to use for classes is also up to your own preference, but in this course, we want to follow the CamelCaseNaming convention.

Like this:

| **class** **MyAwesomeClass**: |
| --- |

### **7.3 Does a user-defined class represent each group of objects that is used in two or more tasks and forms a single conceptual object?**

A single user-defined class should represent a single conceptual object, or idea, which can then be used in multiple tasks. For example, a class that represents a single card can then be used to build a deck of cards, shuffled, and dealt to players. Another example could be a class that represents a single book. The book objects created from this class will each have their own information (title, author, contents), and can then be 'shelved' (or put into a list or dictionary) or read.

### **7.4 Is each instance attribute of a user-defined class only used inside the class definition?**

Instance attributes of a class are variables that have the *self.* before the variable name, and they can be accessed anywhere inside the class definition. In a way, they are similar to global variables, but restricted to the scope of the class.

Instance attributes can also be accessed outside of the class; however, this can be risky as you may accidentally make unintended changes that may then break the program. Accessing instance attributes outside of the class should only be done by using getters and setters ([here is a great article that explains getters and setters more in-depth](https://www.geeksforgeeks.org/getter-and-setter-in-python/)).

### **7.5 Does each method in the class satisfy the software quality tests for user-defined functions?**

Methods in a class are just user-defined functions that are now restricted to a certain class of objects, such that in order to access the method, an object of the class the method belongs to is needed. Thus, all methods also need to follow the same software quality test for user-defined function. This includes everything listed in [2.4 User-defined Function Comment](#_ul067to5g2ff), and [6. User-defined Functions](https://docs.google.com/document/d/1AO_jSy6eB4O2GTclKCgyTyOClmBs5LR1xyPUGnMLLOk/edit#heading=h.kc9ps9jsizl6).

## 